

<p style="text-align: center;">РБНФ №1 (опис синтаксису всіма допустимими засобами РБНФ)</p>	<p style="text-align: center;">РБНФ №2 (опис формальної граматики засобами РБНФ)</p>	<p style="text-align: center;">Формальна граматика</p>	<p style="text-align: center;">Формальна граматика з специфікацією lookahead у правилах для LL(2)-аналізатора</p>	<p style="text-align: center;">/* Перевірка РБНФ №1 за допомогою коду (помістити у файл "EBNF_N1.h") */</p>	<p style="text-align: center;">/* Перевірка РБНФ №2 за допомогою коду (помістити у файл "EBNF_N2.h") */</p>
		G = (N, T, P, S)	G = (N, T, P, S)		
		S → program_rule	S → program_rule		
		<pre>N = { program_name, value_type, array_specify, declaration_element, array_specify_optional, other_declaration_ident, declaration, other_declaration_ident_iteration, index_action, unary_operator, unary_operation, binary_operator, binary_action, left_expression, group_expression, index_action_optional, expression, binary_action_iteration, expression_or_cond_block_with_optional_assign, assign_to_right, assign_to_right_optional, if_expression, body_for_true, false_cond_block_without_else, body_for_false, cond_block, false_cond_block_without_else_iteration, body_for_false_optional, cycle_begin_expression, cycle_end_expression, cycle_counter, cycle_counter_lr_init, cycle_counter_init, cycle_counter_last_value, cycle_body, fordownto_cycle, statement, statement_or_block_statements, block_statements, input_rule, argument_for_input, output_rule, statement_iteration, expression_optional, program_rule, declaration_optional, non_zero_digit, digit_iteration, digit, unsigned_value, value, sign_optional, sign, ident, letter_in_upper_case, letter_in_lower_case, sign_plus, sign_minus }</pre>	<pre>N = { program_name, value_type, array_specify, declaration_element, array_specify_optional, other_declaration_ident, declaration, other_declaration_ident_iteration, index_action, unary_operator, unary_operation, binary_operator, binary_action, left_expression, group_expression, index_action_optional, expression, binary_action_iteration, expression_or_cond_block_with_optional_assign, assign_to_right, assign_to_right_optional, if_expression, body_for_true, false_cond_block_without_else, body_for_false, cond_block, false_cond_block_without_else_iteration, body_for_false_optional, cycle_begin_expression, cycle_end_expression, cycle_counter, cycle_counter_lr_init, cycle_counter_init, cycle_counter_last_value, cycle_body, fordownto_cycle, statement, statement_or_block_statements, block_statements, input_rule, argument_for_input, output_rule, statement_iteration, expression_optional, program_rule, declaration_optional, non_zero_digit, digit_iteration, digit, unsigned_value, value, sign_optional, sign, ident, letter_in_upper_case, letter_in_lower_case, sign_plus, sign_minus }</pre>	<pre>#define NONTERMINALS program_name, \ value_type, \ array_specify, \ declaration_element, \ array_specify_optional, \ other_declaration_ident, \ declaration, \ other_declaration_ident_iteration, \ index_action, \ unary_operator, \ unary_operation, \ binary_operator, \ binary_action, \ left_expression, \ group_expression, \ index_action_optional, \ expression, \ binary_action_iteration, \ expression_or_cond_block_with_optional_assign, \ assign_to_right, \ assign_to_right_optional, \ if_expression, \ body_for_true, \ false_cond_block_without_else, \ body_for_false, \ cond_block, \ false_cond_block_without_else_iteration, \ body_for_false_optional, \ cycle_begin_expression, \ cycle_end_expression, \ cycle_counter, \ cycle_counter_lr_init, \ cycle_counter_init, \ cycle_counter_last_value, \ cycle_body, \ fordownto_cycle, \ statement, \ statement_or_block_statements, \ block_statements, \ input_rule, \ argument_for_input, \ output_rule, \ statement_iteration, \ expression_optional, \ program_rule, \ declaration_optional, \ non_zero_digit, \ digit_iteration, \ digit, \ unsigned_value, \ value, \ sign, \ ident, \ letter_in_upper_case, \ letter_in_lower_case, \ sign_plus, \ sign_minus }</pre>	<pre>#define NONTERMINALS program_name, \ value_type, \ array_specify, \ declaration_element, \ array_specify_optional, \ other_declaration_ident, \ declaration, \ other_declaration_ident_iteration, \ index_action, \ unary_operator, \ unary_operation, \ binary_operator, \ binary_action, \ left_expression, \ group_expression, \ index_action_optional, \ expression, \ binary_action_iteration, \ expression_or_cond_block_with_optional_assign, \ assign_to_right, \ assign_to_right_optional, \ if_expression, \ body_for_true, \ false_cond_block_without_else, \ body_for_false, \ cond_block, \ false_cond_block_without_else_iteration, \ body_for_false_optional, \ cycle_begin_expression, \ cycle_end_expression, \ cycle_counter, \ cycle_counter_lr_init, \ cycle_counter_init, \ cycle_counter_last_value, \ cycle_body, \ fordownto_cycle, \ statement, \ statement_or_block_statements, \ block_statements, \ input_rule, \ argument_for_input, \ output_rule, \ statement_iteration, \ expression_optional, \ program_rule, \ declaration_optional, \ non_zero_digit, \ digit_iteration, \ digit, \ unsigned_value, \ value, \ sign_optional, \ sign, \ ident, \ letter_in_upper_case, \ letter_in_lower_case, \ sign_plus, \ sign_minus }</pre>
		T = { "INTEGER16", ",", "NOT", "AND", "OR", "==", "!=", "<", ">", "+", "-", "*", "/", "DIV", "MOD", "	T = { "LONG", "INT", ",", "NOT", "AND", " ", "EQ", "NE", "<", ">", "+", "-", "ADD", "SUB", "MUL", "DIV", "	<pre>#define TOKENS \ tokenLONG, \ tokenINT, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \</pre>	<pre>#define TOKENS \ tokenLONG, \ tokenINT, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \</pre>

						#define T_RIGHT_SQUAREBRACKETS_3 ""
				tokenBEGINBLOCK = "!" >> BOUNDARIES;	tokenBEGINBLOCK = "!" >> BOUNDARIES;	#define T_BEGIN_BLOCK_0 "" #define T_BEGIN_BLOCK_1 "" #define T_BEGIN_BLOCK_2 "" #define T_BEGIN_BLOCK_3 ""
				tokenENDBLOCK = ")" >> BOUNDARIES;	tokenENDBLOCK = ")" >> BOUNDARIES;	#define T_END_BLOCK_0 "")" #define T_END_BLOCK_1 "") #define T_END_BLOCK_2 "") #define T_END_BLOCK_3 "")
				tokenSEMICOLON = ";" >> BOUNDARIES;	tokenSEMICOLON = ";" >> BOUNDARIES;	#define T_SEMICOLON_0 ";" #define T_SEMICOLON_1 ";" #define T_SEMICOLON_2 ";" #define T_SEMICOLON_3 ";"
				tokenLONG = "LONG" >> STRICT_BOUNDARIES; tokenINT = "INT" >> STRICT_BOUNDARIES;	tokenLONG = "LONG" >> STRICT_BOUNDARIES; tokenINT = "INT" >> STRICT_BOUNDARIES;	#define T_DATA_TYPE_0 "LONG" #define T_DATA_TYPE_1 "INT" #define T_DATA_TYPE_2 "" #define T_DATA_TYPE_3 ""
				tokenCOMMA = "," >> BOUNDARIES;	tokenCOMMA = "," >> BOUNDARIES;	#define T_COMA_0 "," #define T_COMA_1 "," #define T_COMA_2 "," #define T_COMA_3 ","
						#define T_BITWISE_NOT_0 "~" #define T_BITWISE_NOT_1 "" #define T_BITWISE_NOT_2 "" #define T_BITWISE_NOT_3 ""
				tokenNOT = "NOT" >> STRICT_BOUNDARIES;	tokenNOT = "NOT" >> STRICT_BOUNDARIES;	#define T_NOT_0 "NOT" #define T_NOT_1 "" #define T_NOT_2 "" #define T_NOT_3 ""
						#define T_BITWISE_AND_0 "&" #define T_BITWISE_AND_1 "" #define T_BITWISE_AND_2 "" #define T_BITWISE_AND_3 ""
				tokenAND = "AND" >> STRICT_BOUNDARIES;	tokenAND = "AND" >> STRICT_BOUNDARIES;	#define T_AND_0 "AND" #define T_AND_1 "" #define T_AND_2 "" #define T_AND_3 ""
						#define T_BITWISE_OR_0 " " #define T_BITWISE_OR_1 "" #define T_BITWISE_OR_2 "" #define T_BITWISE_OR_3 ""
				tokenOR = " " >> STRICT_BOUNDARIES;	tokenOR = " " >> STRICT_BOUNDARIES;	#define T_OR_0 " " #define T_OR_1 "" #define T_OR_2 "" #define T_OR_3 ""
				tokenEQUAL = "EQ" >> BOUNDARIES;	tokenEQUAL = "EQ" >> BOUNDARIES;	#define T_EQUAL_0 "EQ" #define T_EQUAL_1 "" #define T_EQUAL_2 "" #define T_EQUAL_3 ""
				tokenNOTEQUAL = "NE" >> BOUNDARIES;	tokenNOTEQUAL = "NE" >> BOUNDARIES;	#define T_NOT_EQUAL_0 "NE" #define T_NOT_EQUAL_1 "" #define T_NOT_EQUAL_2 "" #define T_NOT_EQUAL_3 ""
				tokenLESS = "<" >> BOUNDARIES;	tokenLESS = "<" >> BOUNDARIES;	#define T_LESS_0 "<" #define T_LESS_1 "" #define T_LESS_2 "" #define T_LESS_3 ""
				tokenGREATER = ">" >> BOUNDARIES;	tokenGREATER = ">" >> BOUNDARIES;	#define T_GREATER_0 ">" #define T_GREATER_1 "" #define T_GREATER_2 "" #define T_GREATER_3 ""
				tokenPLUS = "ADD" >> BOUNDARIES;	tokenPLUS = "ADD" >> BOUNDARIES;	#define T_ADD_0 "ADD" #define T_ADD_1 "" #define T_ADD_2 "" #define T_ADD_3 ""
				tokenMINUS = "SUB" >> BOUNDARIES;	tokenMINUS = "SUB" >> BOUNDARIES;	#define T_SUB_0 "SUB" #define T_SUB_1 "" #define T_SUB_2 "" #define T_SUB_3 ""
				tokenMUL = "MUL" >> BOUNDARIES;	tokenMUL = "MUL" >> BOUNDARIES;	#define T_MUL_0 "MUL" #define T_MUL_1 "" #define T_MUL_2 "" #define T_MUL_3 ""
				tokenDIV = "DIV" >> STRICT_BOUNDARIES;	tokenDIV = "DIV" >> STRICT_BOUNDARIES;	#define T_DIV_0 "DIV" #define T_DIV_1 "" #define T_DIV_2 "" #define T_DIV_3 ""
				tokenMOD = "MOD" >> STRICT_BOUNDARIES;	tokenMOD = "MOD" >> STRICT_BOUNDARIES;	#define T_MOD_0 "MOD" #define T_MOD_1 "" #define T_MOD_2 "" #define T_MOD_3 ""
				tokenLRASSIGN = ":" >> BOUNDARIES;	tokenLRASSIGN = ":" >> BOUNDARIES;	#define T_LRASSIGN_0 ":">" #define T_LRASSIGN_1 "" #define T_LRASSIGN_2 "" #define T_LRASSIGN_3 ""
						#define T_THEN_BLOCK_0 "(" #define T_THEN_BLOCK_1 ")" #define T_THEN_BLOCK_2 ")" #define T_THEN_BLOCK_3 ")"
				tokenELSE = "ELSE" >> STRICT_BOUNDARIES;	tokenELSE = "ELSE" >> STRICT_BOUNDARIES;	#define T_ELSE_BLOCK_0 "ELSE" #define T_ELSE_BLOCK_1 T_BEGIN_BLOCK_0 #define T_ELSE_BLOCK_2 "" #define T_ELSE_BLOCK_3 ""
				tokenIF = "IF" >> STRICT_BOUNDARIES;	tokenIF = "IF" >> STRICT_BOUNDARIES;	#define T_IF_0 "IF" #define T_IF_1 "" #define T_IF_2 "" #define T_IF_3 ""
				tokenDO = "DO" >> STRICT_BOUNDARIES;	tokenDO = "DO" >> STRICT_BOUNDARIES;	#define T_DO_0 "DO" #define T_DO_1 "" #define T_DO_2 "" #define T_DO_3 ""
				tokenFOR = "FOR" >> STRICT_BOUNDARIES;	tokenFOR = "FOR" >> STRICT_BOUNDARIES;	#define T_FOR_0 "FOR"

						})\n{ LA_IS, { T_SUB_0 }, { "binary_operator",{\n{ LA_IS, { "" }, 1, { T_SUB_0 } }\n}}\n{ LA_IS, { T_MUL_0 }, { "binary_operator",{\n{ LA_IS, { "" }, 1, { T_MUL_0 } }\n}}\n{ LA_IS, { T_DIV_0 }, { "binary_operator",{\n{ LA_IS, { "" }, 1, { T_DIV_0 } }\n}}\n{ LA_IS, { T_MOD_0 }, { "binary_operator",{\n{ LA_IS, { "" }, 1, { T_MOD_0 } }\n}}\n})\n
binary_action = binary_operator , expression;	binary_action = binary_operator , expression;	binary_action → binary_operator expression	binary_action(1: "AND", " ", "EQ", "NE", "<", ">", "ADD", "SUB", "MUL", "DIV", "MOD") → binary_operator expression	binary_action = binary_operator >> expression;	binary_action = binary_operator >> expression;	{ LA_IS, { T_AND_0, T_OR_0, T_EQUAL_0, T_NOT_EQUAL_0, T_LESS_0, T_GREATER_0, T_ADD_0, T_SUB_0, T_MUL_0, T_DIV_0, T_MOD_0 }, { "binary_action",{\n{ LA_IS, { "" }, 2, { "binary_operator", "expression" } }\n}}\n
left_expression = group_expression unary_operation cond_block value ident , [index_action];	left_expression = group_expression unary_operation cond_block value ident , [index_action];	left_expression → group_expression unary_operation cond_block value ident , index_action_optional;	left_expression(1: "(") → group_expression\nleft_expression(1: "NOT") → unary_operation\nleft_expression(1: "IF") → cond_block\nleft_expression → value\nleft_expression → ident , index_action_optional	left_expression(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → value\nleft_expression(1: "ADD", "SUB", "2: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → value\nleft_expression(1: "_") → ident , index_action_optional	left_expression = group_expression unary_operation cond_block value ident >> -index_action;	{ LA_IS, { "" }, { "left_expression",{\n{ LA_IS, { "" }, 1, { "group_expression" } }\n}}\n{ LA_IS, { T_NOT_0 }, { "left_expression",{\n{ LA_IS, { "" }, 1, { "unary_operation" } }\n}}\n{ LA_IS, { T_IF_0 }, { "left_expression",{\n{ LA_IS, { "" }, 1, { "cond_block" } }\n}}\n{ LA_IS, { "unsigned_value_terminal" }, { "left_expression",{\n{ LA_IS, { "" }, 1, { "value" } }\n}}\n{ LA_IS, { T_ADD_0, T_SUB_0 }, { "left_expression",{\n{ LA_IS, { "unsigned_value_terminal" }, 1, { "value" } }\n}}\n{ LA_IS, { "LA_NOT", { "unsigned_value_terminal" }, 1, { "unary_operation" } }\n{ LA_IS, { "ident_terminal" }, { "left_expression",{\n{ LA_IS, { "" }, 2, { "ident", "index_action_optional" } }\n}}\n})\n
expression = left_expression , {binary_action};	expression = left_expression , {binary_action};	index_action_optional = index_action ε;	index_action_optional → index_action\nindex_action_optional → ε	index_action_optional(1: "[") → index_action\nindex_action_optional(1: "![") → ε		index_action_optional = index_action "";\n{ LA_IS, { "" }, { "index_action_optional",{\n{ LA_IS, { "" }, 1, { "index_action" } }\n}}\n{ LA_NOT, { "[" }, { "index_action_optional",{\n{ LA_IS, { "" }, 0, { "}" } }\n}}\n})\n
group_expression = (" , expression , ") ;	group_expression = (" , expression , ") ;	expression → left_expression	expression(1: "NOT", "ADD", "SUB", "_", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → left_expression binary_action_iteration	expression = left_expression >> *binary_action;	expression = left_expression >> binary_action_iteration;	{ LA_IS, { "(", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "expression",{\n{ LA_IS, { "" }, 2, { "left_expression", "binary_action_iteration" } }\n}}\n})\n
expression_or_cond_block_with_optional_assign = expression , [">"] , ident , [index_action];	expression_or_cond_block_with_optional_assign = expression , [">"] , ident , [index_action];	binary_action_iteration = binary_action, binary_action_iteration ε;	binary_action_iteration → binary_action\nbinary_action_iteration → ε	binary_action_iteration(1: "AND", " ", "EQ", "NE", "<", ">", "ADD", "SUB", "MUL", "DIV", "MOD") → binary_action binary_action_iteration\nbinary_action_iteration(1: "AND", "!", " ", "EQ", "NE", "<", ">", "ADD", "SUB", "MUL", "DIV", "MOD") → ε		{ LA_IS, { T_AND_0, T_OR_0, T_EQUAL_0, T_NOT_EQUAL_0, T_LESS_0, T_GREATER_0, T_ADD_0, T_SUB_0, T_MUL_0, T_DIV_0, T_MOD_0 }, { "binary_action_iteration",{\n{ LA_IS, { "" }, 2, { "binary_action", "binary_action_iteration" } }\n}}\n{ LA_NOT, { T_AND_0, T_OR_0, T_EQUAL_0, T_NOT_EQUAL_0, T_LESS_0, T_GREATER_0, T_ADD_0, T_SUB_0, T_MUL_0, T_DIV_0, T_MOD_0 }, { "binary_action_iteration",{\n{ LA_IS, { "" }, 0, { "}" } }\n}}\n})\n
if_expression = expression;	if_expression = expression;	group_expression = "(" , expression , ")"	group_expression(1: "(") → "(" expression ")"	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;	{ LA_IS, { "" }, { "group_expression",{\n{ LA_IS, { "" }, 3, { "(", "expression", ")" } }\n}}\n})\n
body_for_true = block_statements;	body_for_true = block_statements;	expression_or_cond_block_with_optional_assign = expression , assign_to_right_optional;	assign_to_right_optional → assign_to_right ε;	expression_or_cond_block_with_optional_assign(1: "NOT", "ADD", "SUB", "_", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression\nassign_to_right_optional	expression_or_cond_block_with_optional_assign = expression >> -(tokenLRASSIGN>> ident >> -index_action);	expression_or_cond_block_with_optional_assign = expression >> assign_to_right_optional;\n{ LA_IS, { "(", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "expression_or_cond_block_with_optional_assign",{\n{ LA_IS, { "" }, 2, { "expression", "assign_to_right_optional" } }\n}}\n})\n
false_cond_block_without_else = "ELSE" , "IF" , if_expression , body_for_true;	false_cond_block_without_else = "ELSE" , "IF" , if_expression , body_for_true;	if_expression = expression;	if_expression → expression	if_expression(1: "NOT", "ADD", "SUB", "_", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression	if_expression = SAME_RULE(expression);	{ LA_IS, { "(", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "if_expression",{\n{ LA_IS, { "" }, 1, { "expression" } }\n}}\n})\n
body_for_false = "ELSE" , block_statements;	body_for_false = "ELSE" , block_statements;	body_for_false = "ELSE" , block_statements;	body_for_false → "ELSE" block_statements	body_for_false(1: "ELSE") → "ELSE" block_statements	body_for_true = SAME_RULE(block_statements);	{ LA_IS, { T_BEGIN_BLOCK_0 }, { "body_for_true",{\n{ LA_IS, { "" }, 1, { "block_statements" } }\n}}\n})\n
cond_block = "IF" , if_expression , body_for_true , false_cond_block_without_else_iteration , body_for_false_optional;	cond_block = "IF" , if_expression , body_for_true , false_cond_block_without_else_iteration , body_for_false_optional;	false_cond_block_without_else = "ELSE" "IF" if_expression body_for_true;	false_cond_block_without_else(1: "ELSE") → "ELSE" "IF" if_expression body_for_true	false_cond_block_without_else = tokenELSE >> tokenIF >> if_expression >> body_for_true;	body_for_false = tokenELSE >> block_statements;	{ LA_IS, { T_ELSE_IF_0 }, { "false_cond_block_without_else",{\n{ LA_IS, { "" }, 4, { T_ELSE_IF_0, T_ELSE_IF_1, "if_expression", "body_for_true" } }\n}}\n})\n
				cond_block(1: "IF") → "IF" if_expression\nbody_for_true >> false_cond_block_without_else_iteration\nbody_for_false_optional	body_for_false = tokenELSE >> block_statements;	{ LA_IS, { T_ELSE_BLOCK_0 }, { "body_for_false",{\n{ LA_IS, { "" }, 2, { T_ELSE_BLOCK_0, "block_statements" } }\n}}\n})\n
				cond_block = tokenIF >> if_expression >> body_for_true >> *false_cond_block_without_else_iteration >> body_for_false;	cond_block = tokenIF >> if_expression >> body_for_true >> false_cond_block_without_else_iteration >> body_for_false_optional;	{ LA_IS, { T_IF_0 }, { "cond_block",{\n{ LA_IS, { "" }, 5, { T_IF_0, "if_expression", "body_for_true", "false_cond_block_without_else_iteration", "body_for_false_optional" } }\n}}\n})\n

	false_cond_block_without_else_iteration = false_cond_block_without_else, false_cond_block_without_else_iteration ε;	false_cond_block_without_else_iteration → false_cond_block_without_else_iteration(1: "ELSE", 2: "IF") → false_cond_block_without_else false_cond_block_without_else_iteration false_cond_block_without_else_iteration → ε false_cond_block_without_else_iteration(1: "ELSE", 2: !"IF") → ε false_cond_block_without_else_iteration(1: !"ELSE") → ε	false_cond_block_without_else_iteration(1: "ELSE", 2: "IF") → false_cond_block_without_else false_cond_block_without_else_iteration false_cond_block_without_else_iteration(1: "ELSE", 2: !"IF") → ε false_cond_block_without_else_iteration(1: !"ELSE") → ε		false_cond_block_without_else_iteration = false_cond_block_without_else>> false_cond_block_without_else_iteration "";	{LA_IS, {T_ELSE_IF_0}, {"false_cond_block_without_else_iteration"}, {LA_IS, {T_ELSE_IF_1}}, 2, {"false_cond_block_without_else", "false_cond_block_without_else_iteration"}}, {LA_NOT, {T_ELSE_IF_1}, 0, {"""}}}}
	body_for_false_optional = body_for_false ε;	body_for_false_optional → body_for_false body_for_false_optional → ε	body_for_false_optional(1: "FALSE") → body_for_false body_for_false_optional(1: !"FALSE") → ε		body_for_false_optional = body_for_false "";	{LA_IS, {T_ELSE_BLOCK_0}, {"body_for_false_optional"}, {LA_IS, {""}}, 1, {"body_for_false"}}, {LA_NOT, {T_ELSE_BLOCK_0}, {"body_for_false_optional"}, {LA_IS, {""}}, 0, {"""}}}}
cycle_begin_expression = expression;	cycle_begin_expression = expression;	cycle_begin_expression → expression	cycle_begin_expression(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression	cycle_begin_expression = SAME_RULE(expression);	cycle_begin_expression = SAME_RULE(expression);	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal"}, T_IF_0}, {"cycle_begin_expression"}, {LA_IS, {""}}, 1, {"expression"}}
cycle_end_expression = expression;	cycle_end_expression = expression;	cycle_end_expression → expression	cycle_end_expression(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression	cycle_end_expression = SAME_RULE(expression);	cycle_end_expression = SAME_RULE(expression);	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal"}, T_IF_0}, {"cycle_end_expression"}, {LA_IS, {""}}, 1, {"expression"}}
cycle_counter = ident;	cycle_counter = ident;	cycle_counter → ident	cycle_counter(1: "_") → ident	cycle_counter = SAME_RULE(ident);	cycle_counter = SAME_RULE(ident);	{LA_IS, {"ident_terminal"}, {"cycle_counter"}, {LA_IS, {""}}, 1, {"ident"}}
cycle_counter_lr_init = cycle_begin_expression, ">", cycle_counter;	cycle_counter_lr_init = cycle_begin_expression, ">", cycle_counter;	cycle_counter_lr_init → cycle_begin_expression, ">" cycle_counter	cycle_counter_lr_init(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → cycle_begin_expression ":" cycle_counter	cycle_counter_lr_init = cycle_begin_expression >> tokenLRASSIGN >> cycle_counter;	cycle_counter_lr_init = cycle_begin_expression >> tokenLRASSIGN >> cycle_counter;	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal"}, T_IF_0}, {"cycle_counter_lr_init"}, {LA_IS, {""}}, 3, {"cycle_begin_expression", T_LRASSIGN_0, "cycle_counter"}}
cycle_counter_init = cycle_counter_lr_init;	cycle_counter_init = cycle_counter_lr_init;	cycle_counter_init → cycle_counter_lr_init	cycle_counter_init(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → cycle_counter_lr_init	cycle_counter_init = SAME_RULE(cycle_counter_lr_init);	cycle_counter_init = SAME_RULE(cycle_counter_lr_init);	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal"}, T_IF_0}, {"cycle_counter_init"}, {LA_IS, {""}}, 1, {"cycle_counter_lr_init"}}
cycle_counter_last_value = cycle_end_expression;	cycle_counter_last_value = cycle_end_expression;	cycle_counter_last_value → cycle_end_expression	cycle_counter_last_value(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → cycle_end_expression	cycle_counter_last_value = SAME_RULE(cycle_end_expression);	cycle_counter_last_value = SAME_RULE(cycle_end_expression);	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal"}, T_IF_0}, {"cycle_counter_last_value"}, {LA_IS, {""}}, 1, {"cycle_end_expression"}}
cycle_body = "DO", statement_or_block_statements;	cycle_body = "DO", statement_or_block_statements;	cycle_body → "DO" statement_or_block_statements	cycle_body(1: "DO") → "DO" statement_or_block_statements	cycle_body = tokenDO >> (statement block_statements);	cycle_body = tokenDO >> statement_or_block_statements;	{LA_IS, {T_DO_0}, {"cycle_body"}, {LA_IS, {""}}, 2, {T_DO_0, "statement_or_block_statements"}}
fordownto_cycle = "FOR", cycle_counter_init, "DOWNTO", cycle_counter_last_value, cycle_body;	fordownto_cycle = "FOR", cycle_counter_init, "DOWNTO", cycle_counter_last_value cycle_body;	fordownto_cycle → "FOR" cycle_counter_init "DOWNTO", cycle_counter_last_value cycle_body	fordownto_cycle(1: "FOR") → "FOR" cycle_counter_init "DOWNTO", cycle_counter_last_value cycle_body	fordownto_cycle = tokenFOR >> cycle_counter_init >> tokenDOWNTO >> cycle_counter_last_value >> cycle_body;	fordownto_cycle = tokenFOR >> cycle_counter_init >> tokenDOWNTO >> cycle_counter_last_value >> cycle_body;	{LA_IS, {T_FOR_0}, {"fordownto_cycle"}, {LA_IS, {""}}, 5, {T_FOR_0, "cycle_counter_init", T_DOWNTO_0, "cycle_counter_last_value", "cycle_body"}}
statement_or_block_statements = statement block_statements;	statement_or_block_statements = statement block_statements;	statement_or_block_statements → statement block_statements	statement_or_block_statements(1: !"") → statement statement_or_block_statements(1: "") → block_statements		statement_or_block_statements = statement block_statements;	{LA_IS, {"ident_terminal", "", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_FOR_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0}, {"statement_or_block_statements"}, {LA_IS, {""}}, 1, {"statement"}}
input_rule = "SCAN", argument_for_input;	input_rule = "SCAN", argument_for_input;	input_rule → "SCAN" argument_for_input	input_rule(1: "SCAN") → "SCAN" argument_for_input	input_rule = tokenGET >> (ident >> -index_action tokenGROUPEXPRESSIONBEGIN >> ident >> -index_action >> tokenGROUPEXPRESSIONEND);	input_rule = tokenGET >> argument_for_input;	{LA_IS, {T_INPUT_0}, {"input_rule"}, {LA_IS, {""}}, 2, {T_INPUT_0, "argument_for_input"}}
input_rule = "SCAN", (ident , [index_action] "(" , ident , [index_action] , ")");	argument_for_input = ident , index_action_optional; argument_for_input = "(" , ident , "index_action_optional" , ")" ;	argument_for_input → ident index_action_optional argument_for_input → "(" , ident , "index_action_optional" , ")" ;	argument_for_input(1: "_") → ident index_action_optional argument_for_input(1: "") → "(" , ident , "index_action_optional" , ")" ;		argument_for_input = ident >> index_action_optional tokenGROUPEXPRESSIONBEGIN >> ident >> index_action_optional >> tokenGROUPEXPRESSIONEND;	{LA_IS, {"ident_terminal"}, {"argument_for_input"}, {LA_IS, {""}}, 2, {"ident", "index_action_optional"}}
output_rule = "PRINT", expression;	output_rule = "PRINT", expression;	output_rule → "PRINT" expression	output(1: "PRINT") → "PRINT" expression	output_rule = tokenPUT >> expression;	output_rule = tokenPUT >> expression;	{LA_IS, {T_OUTPUT_0}, {"output_rule"}, {LA_IS, {""}}, 2, {T_OUTPUT_0, "expression"}}
statement = expression_or_cond_block_with_optional_assign fordownto_cycle input_rule output_rule ";" ;	statement = expression_or_cond_block_with_optional_assign fordownto_cycle input_rule output_rule ";" ;	statement → expression_or_cond_block_with_optional_assign statement → fordownto_cycle statement → input_rule statement → output_rule statement → ";" ;	statement(1: ("", "NOT", "ADD", "SUB", "-", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression_or_cond_block_with_optional_assign statement → fordownto_cycle statement → input_rule statement → output_rule statement → ";" ;		statement = expression_or_cond_block_with_optional_assign fordownto_cycle input_rule output_rule tokenSEMICOLON;	{LA_IS, {"", T_NOT_0, "ident_terminal", "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0}, {"statement"}, {LA_IS, {""}}, 1, {"expression_or_cond_block_with_optional_assign"}}
statement = expression_or_cond_block_with_optional_assign fordownto_cycle input_rule output_rule ";" ;	statement_iteration = statement, statement_iteration ε;	statement_iteration → statement statement_iteration statement_iteration → ε	statement_iteration(1: "", ("", "NOT", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB", "IF", "FOR", "SCAN", "PRINT", "") → statement statement_iteration statement_iteration(1: "", !("NOT", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB", "IF", "FOR", "SCAN", "PRINT", !")) → ε		statement_iteration = statement >> statement_iteration "";	{LA_IS, {"ident_terminal", "", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_FOR_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0}, {"statement_iteration"}, {LA_IS, {""}}, 2, {"statement", "statement_iteration"}}

						"unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_FOR_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0}, {"statement_iteration": {\ LA_IS, ("")}, O, { "" }\\ })}\\
block_statements = "{" , statement, "}" ;	block_statements = "{" , statement__iteration , "}" ;	block_statements → "(" statement__iteration ")"	block_statements(1: "(") → "(" statement__iteration ")"	block_statements = tokenBEGINBLOCK >> *statement >> tokenENDBLOCK;	block_statements = tokenBEGINBLOCK >> statement__iteration >> tokenENDBLOCK;	{ LA_IS, { T_BEGIN_BLOCK_0 }, {"block_statements",{\ LA_IS, ("")}, 3, { T_BEGIN_BLOCK_0, "statement_iteration", T_END_BLOCK_0 } }\\ })}\\
	expression__optional = expression "";	expression__optional → expression expression__optional → ε	expression__optional(1: (" ", "NOT", "ADD", "SUB", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression expression__optional(1: !"(", !"NOT", !"ADD", !"SUB", !"!", !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9", !"IF") → ε		expression__optional = expression "";	{ LA_IS, { ("", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0), {"expression_optional": {\ LA_IS, ("")}, 1, { "expression" } }\\ })}\\ { LA_NOT, { ("", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0), {"expression_optional": {\ LA_IS, ("")}, 0, { "" } }\\ })}\\
program_rule = "PROGRAM" , program_name , ";" , "DATA" , declaration__optional , ";" , "BEGIN" , statement__iteration , "END" ; program_rule = "PROGRAM" , program_name , ";" , "DATA" , [declaration] , ";" , "BEGIN" , {statement} , "END" ;	program_rule → "PROGRAM" program_name ";" , "DATA" , declaration__optional ";" , "BEGIN" , statement__iteration "END"	program_rule(1: "PROGRAM") → "PROGRAM" program_name → "DATA" , declaration__optional ";" , "BEGIN" , statement__iteration "END"	program_rule(1: "PROGRAM") → "PROGRAM" program_name → "DATA" , declaration__optional ";" , "BEGIN" , statement__iteration "END"	program_rule = BOUNDARIES >> tokenNAME >> program_name >> tokenSEMICOLON >> tokenDATA >> (-declaration) >> tokenSEMICOLON >> tokenBEGIN >> *statement >> tokenEND;	program_rule = BOUNDARIES >> tokenNAME >> program_name >> tokenSEMICOLON >> tokenDATA >> (-declaration) >> tokenSEMICOLON >> tokenBEGIN >> *statement >> tokenEND;	{ LA_IS, { T_NAME_0 }, {"program_rule",{\ LA_IS, ("")}, 9, { T_NAME_0, "program_name", T_SEMICOLON_0, T_DATA_0, "declaration_optional", T_SEMICOLON_0, T_BEGIN_0, "statement_iteration", T_END_0 } }\\ })}\\
value = [sign] , unsigned_value;	declaration__optional = declaration "";	declaration__optional → declaration declaration__optional → ε	declaration__optional(1: "LONG") → declaration declaration__optional(1: !"LONG") → ε		declaration__optional = declaration "";	{ LA_IS, { T_DATA_TYPE_0 }, {"declaration_optional",{\ LA_IS, ("")}, 1, { "declaration" } }\\ })}\\ { LA_NOT, { T_DATA_TYPE_0 }, {"declaration_optional",{\ LA_IS, ("")}, 0, { "" } }\\ })}\\
	value = sign__optional , unsigned_value;	value → sign__optional unsigned_value	value(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB") → sign__optional unsigned_value		value = sign__optional >> unsigned_value >> BOUNDARIES;	{ LA_IS, { "unsigned_value_terminal", T_ADD_0, T_SUB_0 }, {"value",{\ LA_IS, ("")}, 2, {"sign_optional", "unsigned_value" } }\\ })}\\
	sign__optional = sign ε;	sign__optional → sign sign__optional → ε	sign__optional(1: "ADD", "SUB") → sign sign__optional(1: !"ADD", !"") → ε		sign__optional = sign "";	{ LA_IS, { T_ADD_0, T_SUB_0 }, {"sign_optional",{\ LA_IS, ("")}, 1, {"sign" } }\\ })}\\ { LA_NOT, { T_ADD_0, T_SUB_0 }, {"sign_optional",{\ LA_IS, ("")}, 0, {""" } }\\ })}\\
sign = sign_plus sign_minus;	sign = sign_plus sign_minus;	sign → sign_plus sign → sign_minus	sign(1: "ADD") → sign_plus sign(1: "SUB") → sign_minus		sign = sign_plus sign_minus;	{ LA_IS, { T_ADD_0 }, {"sign",{\ LA_IS, ("")}, 1, {"sign_plus" } }\\ })}\\ { LA_IS, { T_SUB_0 }, {"sign",{\ LA_IS, ("")}, 1, {"sign_minus" } }\\ })}\\
sign_plus = "ADD";	sign_plus = "ADD";	sign_plus → "ADD"	sign_plus(1: "ADD") → "ADD"	sign_plus = SAME_RULE(tokenPLUS);	sign_plus = SAME_RULE(tokenPLUS);	{ LA_IS, { T_ADD_0 }, {"sign_plus",{\ LA_IS, ("")}, 1, { T_ADD_0 } }\\ })}\\
sign_minus = "SUB";	sign_minus = "SUB";	sign_minus → "SUB"	sign_minus(1: "SUB") → "SUB"	sign_minus = SAME_RULE(tokenMINUS);	sign_minus = SAME_RULE(tokenMINUS);	{ LA_IS, { T_SUB_0 }, {"sign_minus",{\ LA_IS, ("")}, 1, { T_SUB_0 } }\\ })}\\
unsigned_value = non_zero_digit , digit__iteration "0";	unsigned_value = non_zero_digit , digit__iteration "0";	unsigned_value → non_zero_digit digit__iteration "0"; unsigned_value → "0"	unsigned_value(1: "1", "2", "3", "4", "5", "6", "7", "8", "9") → non_zero_digit digit__iteration unsigned_value(1: "0") → "0"	unsigned_value = (non_zero_digit >> *digit digit_0) >> BOUNDARIES;	unsigned_value = (non_zero_digit >> digit_0) >> BOUNDARIES;	/* unsigned_value token represents unsigned_value in lexical analyzer */\n{ LA_IS, { "unsigned_value_terminal" }, {"unsigned_value",{\ LA_IS, ("")}, 1, {"unsigned_value_terminal" } }\\ })}\\
	digit__iteration = digit , digit__iteration ε;	digit__iteration → digit digit__iteration digit__iteration → ε	digit__iteration(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → digit digit__iteration digit__iteration(1: !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9") → ε		digit__iteration = digit >> digit__iteration "";	\
digit = "0" non_zero_digit;	digit = "0" non_zero_digit;	digit → "0" digit → non_zero_digit	digit(1: "0") → "0" digit(1: "1", "2", "3", "4", "5", "6", "7", "8", "9") → non_zero_digit	digit_0 = '0'; digit = digit_0 non_zero_digit;	digit_0 = '0'; digit = digit_0 non_zero_digit;	\
non_zero_digit = "1" "2" "3" "4" "5" "6" "7" "8" "9";	non_zero_digit = "1" "2" "3" "4" "5" "6" "7" "8" "9";	non_zero_digit → "1" non_zero_digit → "2" non_zero_digit → "3" non_zero_digit → "4" non_zero_digit → "5" non_zero_digit → "6" non_zero_digit → "7" non_zero_digit → "8" non_zero_digit → "9"	non_zero_digit(1: "1") → "1" non_zero_digit(1: "2") → "2" non_zero_digit(1: "3") → "3" non_zero_digit(1: "4") → "4" non_zero_digit(1: "5") → "5" non_zero_digit(1: "6") → "6" non_zero_digit(1: "7") → "7" non_zero_digit(1: "8") → "8" non_zero_digit(1: "9") → "9"	digit_1 = '1'; digit_2 = '2'; digit_3 = '3'; digit_4 = '4'; digit_5 = '5'; digit_6 = '6'; digit_7 = '7'; digit_8 = '8'; digit_9 = '9'; non_zero_digit = digit_1 digit_2 digit_3 digit_4 digit_5 digit_6 digit_7 digit_8 digit_9;	digit_1 = '1'; digit_2 = '2'; digit_3 = '3'; digit_4 = '4'; digit_5 = '5'; digit_6 = '6'; digit_7 = '7'; digit_8 = '8'; digit_9 = '9'; non_zero_digit = digit_1 digit_2 digit_3 digit_4 digit_5 digit_6 digit_7 digit_8 digit_9;	\
ident = "_" , letter_in_upper_case , letter_in_upper_case , digit;	ident → "_" letter_in_upper_case letter_in_upper_case digit	ident(1: "_") → "_" letter_in_upper_case letter_in_upper_case digit	identUNDERSCORE = "_"; ident = {\ tokenLONG tokenINT tokenCOMMA tokenNOT tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenNOTEQUAL tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD tokenGROUPEXPRESSIONBEGIN tokenGROUPEXPRESSIONEND tokenLASSIGN tokenELSE tokenIF tokenDO tokenFOR tokenDOWNTO tokenGET tokenPUT tokenNAME	identUNDERSCORE = "_"; ident = {\ tokenLONG tokenINT tokenCOMMA tokenNOT tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenNOTEQUAL tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD tokenGROUPEXPRESSIONBEGIN tokenGROUPEXPRESSIONEND tokenLASSIGN tokenELSE tokenIF tokenDO tokenFOR tokenDOWNTO tokenGET tokenPUT tokenNAME	/* ident_token represents ident in lexical analyzer */\n{ LA_IS, { "ident_terminal" }, {"ident",{\ LA_IS, ("")}, 1, {"ident_terminal" } }\\ })}\\	
ident = "_" , letter_in_upper_case , letter_in_upper_case , digit;						

				<pre> tokenDATA tokenBEGIN tokenEND tokenBEGINBLOCK tokenENDBLOCK tokenLEFTSQUAREBRACKETS tokenRIGHTSQUAREBRACKETS tokenSEMICOLON) >> tokenUNDERSCORE >> letter_in_upper_case >> letter_in_upper_case >> digit >> STRICT_BOUNDARIES; </pre>	<pre> tokenDATA tokenBEGIN tokenEND tokenBEGINBLOCK tokenENDBLOCK tokenLEFTSQUAREBRACKETS tokenRIGHTSQUAREBRACKETS tokenSEMICOLON) >> tokenUNDERSCORE >> letter_in_upper_case >> letter_in_upper_case >> digit >> STRICT_BOUNDARIES; </pre>	\	
				<pre> letter_in_lower_case = "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"; letter_in_lower_case = "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"; </pre>	<pre> letter_in_lower_case1: "a" → "a" letter_in_lower_case1: "b" → "b" letter_in_lower_case1: "c" → "c" letter_in_lower_case1: "d" → "d" letter_in_lower_case1: "e" → "e" letter_in_lower_case1: "f" → "f" letter_in_lower_case1: "g" → "g" letter_in_lower_case1: "h" → "h" letter_in_lower_case1: "i" → "i" letter_in_lower_case1: "j" → "j" letter_in_lower_case1: "k" → "k" letter_in_lower_case1: "l" → "l" letter_in_lower_case1: "m" → "m" letter_in_lower_case1: "n" → "n" letter_in_lower_case1: "o" → "o" letter_in_lower_case1: "p" → "p" letter_in_lower_case1: "q" → "q" letter_in_lower_case1: "r" → "r" letter_in_lower_case1: "s" → "s" letter_in_lower_case1: "t" → "t" letter_in_lower_case1: "u" → "u" letter_in_lower_case1: "v" → "v" letter_in_lower_case1: "w" → "w" letter_in_lower_case1: "x" → "x" letter_in_lower_case1: "y" → "y" letter_in_lower_case1: "z" → "z" </pre>	<pre> A = "A"; B = "B"; C = "C"; D = "D"; E = "E"; F = "F"; G = "G"; H = "H"; I = "I"; J = "J"; K = "K"; L = "L"; M = "M"; N = "N"; O = "O"; P = "P"; Q = "Q"; R = "R"; S = "S"; T = "T"; U = "U"; V = "V"; W = "W"; X = "X"; Y = "Y"; Z = "Z"; letter_in_lower_case = a b c d e f g h i j k l m n o p q r s t u v w x y z; </pre>	\
				<pre> letter_in_upper_case = "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"; letter_in_upper_case = "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "X" "Y" "Z"; </pre>	<pre> letter_in_upper_case1: "A" → "A" letter_in_upper_case1: "B" → "B" letter_in_upper_case1: "C" → "C" letter_in_upper_case1: "D" → "D" letter_in_upper_case1: "E" → "E" letter_in_upper_case1: "F" → "F" letter_in_upper_case1: "G" → "G" letter_in_upper_case1: "H" → "H" letter_in_upper_case1: "I" → "I" letter_in_upper_case1: "J" → "J" letter_in_upper_case1: "K" → "K" letter_in_upper_case1: "L" → "L" letter_in_upper_case1: "M" → "M" letter_in_upper_case1: "N" → "N" letter_in_upper_case1: "O" → "O" letter_in_upper_case1: "P" → "P" letter_in_upper_case1: "Q" → "Q" letter_in_upper_case1: "R" → "R" letter_in_upper_case1: "S" → "S" letter_in_upper_case1: "T" → "T" letter_in_upper_case1: "U" → "U" letter_in_upper_case1: "V" → "V" letter_in_upper_case1: "W" → "W" letter_in_upper_case1: "X" → "X" letter_in_upper_case1: "Y" → "Y" letter_in_upper_case1: "Z" → "Z" </pre>	<pre> a = "a"; b = "b"; c = "c"; d = "d"; e = "e"; f = "f"; g = "g"; h = "h"; i = "i"; j = "j"; k = "k"; l = "l"; m = "m"; n = "n"; o = "o"; p = "p"; q = "q"; r = "r"; s = "s"; t = "t"; u = "u"; v = "v"; w = "w"; x = "x"; y = "y"; z = "z"; letter_in_upper_case = A B C D E F G H I J K L M N O P Q R S T U V W X Y Z; </pre>	\
					<pre> STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) (!qi::alpha qi::char("_")); BOUNDARIES = (BOUNDARY >> *(BOUNDARY) NO_BOUNDARY); BOUNDARY = BOUNDARY_SPACE BOUNDARY_TAB BOUNDARY_VERTICAL_TAB BOUNDARY_FORM_FEED BOUNDARY_CARRIAGE_RETURN BOUNDARY_LINE_FEED BOUNDARY_NULL; BOUNDARY_SPACE = " "; BOUNDARY_TAB = "\t"; BOUNDARY_VERTICAL_TAB = "\v"; BOUNDARY_FORM_FEED = "\f"; BOUNDARY_CARRIAGE_RETURN = "\r"; BOUNDARY_LINE_FEED = "\n"; BOUNDARY_NULL = "\0"; NO_BOUNDARY = ""; #define WHITESPACES \ STRICT_BOUNDARIES, \ BOUNDARIES, \ BOUNDARY, \ BOUNDARY_SPACE, \ BOUNDARY_TAB, \ BOUNDARY_VERTICAL_TAB, \ BOUNDARY_FORM_FEED, \ BOUNDARY_CARRIAGE_RETURN, \ BOUNDARY_LINE_FEED, \ BOUNDARY_NULL, \ NO_BOUNDARY </pre>	<pre> STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) (!qi::alpha qi::char("_")); BOUNDARIES = (BOUNDARY >> *(BOUNDARY) NO_BOUNDARY); BOUNDARY = BOUNDARY_SPACE BOUNDARY_TAB BOUNDARY_VERTICAL_TAB BOUNDARY_FORM_FEED BOUNDARY_CARRIAGE_RETURN BOUNDARY_LINE_FEED BOUNDARY_NULL; BOUNDARY_SPACE = " "; BOUNDARY_TAB = "\t"; BOUNDARY_VERTICAL_TAB = "\v"; BOUNDARY_FORM_FEED = "\f"; BOUNDARY_CARRIAGE_RETURN = "\r"; BOUNDARY_LINE_FEED = "\n"; BOUNDARY_NULL = "\0"; NO_BOUNDARY = ""; #define WHITESPACES \ STRICT_BOUNDARIES, \ BOUNDARIES, \ BOUNDARY, \ BOUNDARY_SPACE, \ BOUNDARY_TAB, \ BOUNDARY_VERTICAL_TAB, \ BOUNDARY_FORM_FEED, \ BOUNDARY_CARRIAGE_RETURN, \ BOUNDARY_LINE_FEED, \ BOUNDARY_NULL, \ NO_BOUNDARY </pre>	\
						<pre> \ \ \ \ { LA_IS, { T_NAME_0 }, { "program____part1", \ { LA_IS, {"", 7, T_NAME_0, "program_name", \ T_SEMICOLON_0, T_BODY_0, T_DATA_0, \ "declaration_optional", T_SEMICOLON_0 } } } } \ \ \ "program_rule" </pre>	

